IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Patent Application

Applicant(s):	M.R. Betker et al.
Case:	7-1-3-12-5
Serial No.:	10/072,529
Filing Date:	February 8, 2002
Group:	2171
Examiner:	Juliana K. Kang
Title:	Multiprocessor System with Cache-Based Software Breakpoints

---

DECLARATION UNDER 37 C.F.R. §1.131

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

We, the undersigned, hereby declare and state as follows:

1. We are the inventors of the invention described and claimed in the above-referenced U.S. patent application.

2. At least as early as May 16, 2000, we prepared a description of a software breakpoint mechanism for inclusion in a document entitled "Pegasus SuperCore Macro Specification, Rev. 1.1." This document evidences conception of an invention falling within one or more of the claims of the above-referenced patent application. Relevant portions of this document are attached hereto as Exhibit 1.

3. At least as early as September 7, 2000, the invention was reduced to practice by implementing it in software code. Portions of the software code are attached hereto as Exhibit 2.

A computer-log entry indicating a date associated with the Exhibit 2 software code is attached hereto as Exhibit 3.

4. We understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: 3/22/2005

Michael R. Betker

Date: _____

Han Q. Nguyen

Date: _____

Bryan Schlieder

Date: _____

Shaun P. Whalen

Date: _____

Jay P. Wilshire

A computer-log entry indicating a date associated with the Exhibit 2 software code is attached hereto as Exhibit 3.

4. We understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: _____

Michael R. Betker

Date: 3/24/05

Han Q. Nguyen

Date: _____

Bryan Schlieder

Date: _____

Shaun P. Whalen

Date: _____

Jay P. Wilshire

A computer-log entry indicating a date associated with the Exhibit 2 software code is attached hereto as Exhibit 3.

4. We understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: _____

_____

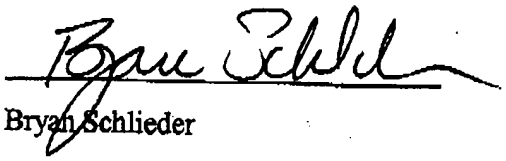
Michael R. Betker

Date: _____

_____

Han Q. Nguyen

Date: 3/22/02



Bryan Schlieder

Date: _____

_____

Shaun P. Whalen

Date: _____

_____

Jay P. Wilshire

A computer-log entry indicating a date associated with the Exhibit 2 software code is attached hereto as Exhibit 3.

4. We understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: _____

_____

Michael R. Betker

Date: _____

_____

Han Q. Nguyen

Date: _____

Bryan Schlieder

Date: 3/23/2005

Shaun P. Whalen

Date: _____

_____

Jay P. Wilshire

A computer-log entry indicating a date associated with the Exhibit 2 software code is attached hereto as Exhibit 3.

4. We understand that willful false statements and the like are punishable by fine or imprisonment, or both, under 18 U.S.C. §1001, and may jeopardize the validity of the application or any patent issuing thereon.

Date: _____ _____

Michael R. Betker

Date: _____ _____

Han Q. Nguyen

Date: _____ _____

Bryan Schlieder

Date: _____ _____

Shaun P. Whalen

Date: 3/23/05 _____

Jay P. Wilshire

Pegasus SuperCore™ Macro Specification

DRAFT

May 16, 2000

Rev 1.1

Table 1 Pegasus Specification Revision History

Revision	Date	Summary
Rev 1.1	*May 16, 2000*	Updated programmers model for separate physical and virtual segments. Updated PIC for 64 interrupts. Updated Daytona bus related sections to match Daytaon Bus spec. 1.1.
Rev 1.0	*March 22, 2000*	First official release.
Rev 0.8	*March 17, 2000*	Added Pegasus Daytona Bus Interface chapter. Changes to Overview, Programmer's Model, Cache Management Unit, Instruction Cache, Data Cache, Write Buffer, Bridge, Daytona Bus Interface, Programmable Interrupt Controller, External Interface.
Rev 0.7	*February 25, 2000*	Changes to Overview, Programmers Model. Added External Interface section.
Rev 0.6	*January 21, 2000*	Changed "SuperCore" to "SuperCore macro".
Rev 0.5	*December 22, 1999*	No change.
Rev 0.4	*December 20, 1999*	Added TM mark.
Rev 0.3	*November 19, 1999*	Changes to Overview, Programmers Model, Cache Manager, ICache, DCache, Write Buffer, Bridge, PIC, Clock Control.
Rev 0.2	*November 5, 1999*	No change.
Rev 0.1	*October 29, 1999*	Initial specification

Table of Contents

1 Overview 1
- 1.1 SuperCore Macro Block Diagram 1
- 1.2 SuperCore Macro Components 3
- 1.3 SuperCore Macro Interfaces 7

2 Programmer's Model 13
- 2.1 SC140 programmer's model 13
- 2.2 Memory map 13
- 2.3 Memory consistency model 23
- 2.4 Cache locking 35
- 2.5 Memory banking 36
- 2.6 Transfer types 37
- 2.7 Execution Stalls 39
- 2.8 Exceptions and Interrupts 40
- 2.9 Control register summary 48

3 Cache Management Unit 55
- 3.1 Overview 55
- 3.2 Local versus Global Memory Regions 56
- 3.3 Global Address Translation 58
- 3.4 External to Local Mapping 61
- 3.5 Local Address Decoding 61
- 3.6 Memory Consistency Models 61
- 3.7 Memory Access Control 64
- 3.8 Exceptions 64
- 3.9 Cache Management Unit Interfaces 64
- 3.10 Registers 65

4 Instruction Cache 77
- 4.1 Organization 77
- 4.2 Cache capacity 80
- 4.3 Cache line size 81
- 4.4 Associativity 81
- 4.5 Supported Access Types 81
- 4.6 Cache invalidation 82
- 4.7 Cache Tags 82
- 4.8 Instruction Cache Internal Structure 83
- 4.9 Preliminary Timing Diagram 84

5 Data Cache 87
- 5.1 Organization 87
- 5.2 Cache line size 88
- 5.3 Associativity 89
- 5.4 Supported Access Types 89
- 5.5 Cache "miss on read" policy 89
- 5.6 Cache "miss on write" policy 90
- 5.7 Cache invalidation 90
- 5.8 Memory banking 90
- 5.9 Data Cache Internal Structure 91

6 Write Buffer 95
- 6.1 Organization 95

6.2 Write buffer depth 96
6.3 Write buffer line size 97
6.4 Supported Access Types 97
6.5 Write Buffer Internal Structure 97
7 Local Data Memory 99
7.1 Organization 100
7.1 Operation 100
7.2 Memory banking 100
7.3 Timing diagrams 101
8 Bridge 103
8.1 Organization 103
8.1 Operation 104
8.2 User-defined LPB Decoder 108
8.3 Errors 109
8.4 Registers 110
8.5 Timing diagrams 110
8.6 Signals 110
9 Daytona Bus Interface 113
9.1 Operation 113
9.2 Reset and Exceptions 124
9.3 Debug 126
9.4 Registers 126
9.5 Timing diagrams 131
10 Programmable Interrupt Controller 135
10.1 Operation 135
10.2 Registers 141
10.3 Signal definitions 151
11 Pegasus Clock Interface 155
11.1 Introduction 155
12 Configuration Interface 157
12.1 Operation 157
12.2 Registers 157
12.3 Timing diagrams 157
12.4 Configuration signals 157
13 JTAG, EOnCE and Scan Chain 161
13.1 Debug/test architecture 161
13.2 JTAG architecture 164
13.3 Debug operations 170
13.4 Debug token bus 174
13.5 Streaming interface 179
13.6 Trace buffer interface 180
13.7 Boundary scan register and user-defined scan chain interface ... 181
13.8 Test 183
13.9 Registers 185
13.10 Timing diagrams 200
13.11 Signals 200
14 External Interface 207
14.1 Pegasus external interface 207
15 Pegasus Daytona Bus Interface 221

4.6 Cache invalidation

Since the SuperCore macro does not support snooping or cache coherency, the processor may need to invalidate some or all of the cache lines under program control. For example, the programmer may construct a program sequence in the global memory and then try to execute it. Also, a debugger program may plant a TRAP opcode in an instruction set. In some cases only a few lines will need to be flushed, in others all lines will be flushed. The instruction cache supports single line and whole array invalidation. A cache line is invalidated by setting the line-valid bit to zero. At the same time the corresponding LRU bit will also be set to 0, indicating that the way-0 line will be over-written first.

4.7 Cache Tags

Each tag in the instruction cache is accessible directly in the local address region of the memory. Tags are accessed as 32-bit values. The instruction cache tags have the format shown in Table 1.

Table 1 Instruction Cache Tag

Bit #	31— 24	23—22	21	20	19—0
Name	R'SVD	SB	L	V	TAG

Bit #	Name	Description
31—24	R'SVD	Reserved.
23—22	SB	Software breakpoint. Indicates the behavior of this cache line when used to assist handling a "software" breakpoint. If 00: normal cache behavior and replacement algorithm; if 01: use_once behavior; if 10: debug_lock behavior; if 11; dont_use_once behavior.
21	L	LRU. The least recently used bit governing which way will be replaced on the next cache miss. There is physically one bit. Reads of this bit in each way report the state of the bit. Writes of this bit in either way update the bit.
20	V	Valid. Indicates whether or not this cache line contains valid data.
19—0	TAG	Address tag. An access to the cache hits this line if the most significant 20 bits of the address match this TAG and the V bit is set and the SB behavior is normal, use_once, or debug_lock.

Multiprocessor software breakpoints on Pegasus are supported by two mechanisms. The two mechanisms are supported by two tag bits in the instruction cache called SB. The first mechanism (referred to as use_once) is intended for use by a target resident debug monitor (or RTOS debug task). This mechanism is relatively intrusive to code running on the SC140s even if that code is not intended to take a software breakpoint. However, there are no hardware restrictions on the number of software breakpoints. The second mechanism (referred to as dont_use_once) operates by locking lines in the cache that include break-

point opcodes. This mechanism is non-intrusive to code that is not intended to take the software breakpoint. However, this mechanism is more restricted as far as the placement of software breakpoints.

The SB bits are encoded as follows:

- 00 - normal instruction cache behavior and replacement algorithm.
- 01 - use_once behavior; service the first fetch of this address from the cache and then invalidate this cache line.
- 10 - debug_lock behavior; service all fetches to this address from the cache and do not allow replacement of this line.
- 11 - dont_use_once behavior; on the first fetch of this address, do not service the access from the cache; after the first fetch, switch behavior to debug_lock; do not change the valid state of this line.

Further information on the use of the software breakpoint bits is found in the JTAG, EOnCE, and Scan section.

4.8 Instruction Cache Internal Structure

The instruction cache contains two single-port SRAM arrays for data and two single-port SRAM arrays for the tags. The valid and LRU bits are implemented separately from the tag since there is a requirement to reset them in a small number of cycles at reset and on a cache flush.

Each data array has 128 rows and 256 columns in a standard format. The write drivers support independent update of the two 16-byte sectors in a cache line. When a cache line is updated the requested data is bypassed to the P-bus pins. The output data must be latched and only change as a result of a processor request.

Each tag array has 128 rows, each row containing a 20-bit tag address field. On read, the tag bits pass directly to 20-bit comparators, that check for equality with the high 20-bits of the request address. If one of the tags matches the address and the valid bit is set, the data for the matching way is forwarded to the SC140.

When a stall situation is detected, the Cache Manager will activate a state-machine to handle the cache fill operation. The timing constraints for these fill cycles will be relatively relaxed.

A rough outline of a two-way set associative implementation is present below for illustration only.

13.2.13 Intest

The extest instruction causes a user-defined boundary scan register to be placed in the scan chain between the TDI and TDO signals. In addition, the user-defined scan chain interface signals are set to indicate test mode for the boundary scan register and that intest is present in the instruction register when this instruction is loaded into the JIR by transiting the Update-IR state.

13.2.14 Bypass instruction

The bypass instruction causes the bypass register flip-flop to be placed in the scan chain between the TDI and TDO signals. In addition, the boundary scan register is placed in the normal mode when this instruction is loaded into the JIR by transiting the Update-IR state. The bypass instruction is also selected if any reserved instruction is loaded into the JIR.

13.2.15 Bypass register

The bypass register is a 1 bit register. On traversing the Capture-DR state, the bypass register is loaded with 0. The bypass register passes its input data to its output signal after being clocked.

13.3 Debug operations

This section describes the JES support for certain common debug operations. The procedure described here is not intended to be the only means for implementing the operation and the debug host/debugger is free to use any means that work.

13.3.1 Software instruction breakpoints

Software instruction breakpoints for a single processor are traditionally implemented by overwriting the instruction at which a breakpoint is desired with a DEBUG or DEBUGEV opcode. This approach does not work if the instruction memory is not writable or in a multiprocessor system. In a multiprocessor system, a software breakpoint may be intended for one or more, but not all, of the CPUs. In this case, a traditional software breakpoint would cause all of the CPUs to stop rather than just the intended one. The JES block addresses these problems with hardware support. This hardware support is in the form of extra tag bits in the instruction cache. These tag bits enable a debug host or target-resident debug monitor to select the level of intrusiveness of "software" breakpoints and sequence a CPU when it encounters such a breakpoint.

Multiprocessor software breakpoints on Pegasus are supported by two mechanisms. The two mechanisms are supported by two tag bits in the instruction cache called swbpt[1:0]. The first mechanism (referred to as use_once) is intended for use by a target resident debug monitor (or RTOS debug task). This mechanism is relatively intrusive to code running on the SC140s even if that code is not intended to take a software breakpoint. How-

ever, there are no hardware restrictions on the number of software breakpoints. The second mechanism (referred to as dont_use_once) operates by locking lines in the cache that include breakpoint opcodes. This mechanism is non-intrusive to code that is not intended to take the software breakpoint. However, this mechanism is more restricted as far as the placement of software breakpoints.

The swbpt bits are encoded as follows:

- 00 - normal instruction cache behavior and replacement algorithm.
- 01 - use_once behavior; service the first fetch of this address from the cache and then invalidate this cache line.
- 10 - debug_lock behavior; service all fetches to this address from the cache and do not allow replacement of this line.
- 11 - dont_use_once behavior; on the first fetch of this address, do not service the access from the cache; after the first fetch, switch behavior to debug_lock; do not change the valid state of this line.

The use_once mechanism would typically be used with a target resident debug monitor or an RTOS debug task although it could also be implemented by a debug host. The basic operation is to set SC140 debug opcodes in shared main memory on instructions that have a software breakpoint in any SC140. Each SC140 takes a debug exception to their debug monitor/debug task when they encounter the debug opcode. The debug exception handler consults a table of software breakpoints in memory for this Pegasus and either gives control to the debugger user or resumes execution depending on if this software breakpoint is intended for this SC140 or not (respectively).

In the case where an SC140 encountered a software breakpoint not intended for it, the swbpt bits makes it possible to "step" the breakpointed instruction in the SC140's instruction cache, even if the instruction is in a noncacheable segment. Strictly speaking, the swbpt bits control whether or not a fetch address is serviced from the instruction cache even if that address is noncacheable. The cache line with the instruction to be stepped is placed in the cache and the swbpt bits set to use_once. Execution is resumed by fetching this address. After this access, the swbpt bits are change to normal cache behavior and that cache line is invalidated. The SC140 may be stalled during this change in the swbpt bits. The swbpt bits are by an LPB write to the instruction cache tag. The cache line with the swbpt bits set for use_once holds a line whose shared main memory copy contains debug opcodes.

The following is an example of the algorithm that may be used by the debug exception handler (and possibly host debugger) to "step" the software breakpoint on SC140s that are not the intended targets of the breakpoint. In the following description, comments such

as // Host: may be implemented by the debugger host or the debug exception handler.

```
// Host: Debug host writes a debug opcode to the shared memory line, using
// a read-modify-write operation.
// Host: Debug host configures eonce for exceptions on debugev opcodes.
debugev exception handler {
      for (i=0; i<size of this core's bpt table; i++) {
            if (!compare(break_address, bpt_table[i].address)) {
      // compare() can use hashing or other comparison operation.
            break;
      }
      break_this_core = FALSE;
      if (bpt_table[i].idmask & this_core's_id) {
            break_this_core = TRUE;
      // exception handler configures eonce for debug mode
      // on debugev opcodes
            DEBUGEV
      // Host: Debug host user has control ...
      // ... resume
      }
      modified_line = *(prog addr *)(break_address)
      modified_line &= bpt_table[i].debug_opcode_mask
      modified_line |= bpt_table[i].opcode
      line_addr = LPB address of cache line for break_address
      *line_addr = modified_line
      if (break_this_core || noncacheable(break_address)) {
            set swbpt bits to use_once
      }
      RTE // from debug exception handler
}
```

This algorithm requires a software breakpoint table for each SC140 in memory. The structure of this table may be whatever is most efficient for the execution of the debug exception handler and use of the application memory. Semaphore access to this table is supported from both code running on the SC140 (through the BMTSET opcode) as well as from the debug host (through the JES block). The access to a specific cache line (based on physical addresses) using a virtual address is supported by the Pegasus address translate register. The target of the RTE is the execution set in the cache line with swbpt set to use_once. If this is not true, the cache line with use_once will prevent other lines with the same cache index from being cached. A cache line with use_once is never replaced by the cache replacement hardware.

The debug lock bit (debug_lock) mechanism is used when a more non-intrusive software breakpoint is required. Setting the swbpt bits to debug_lock indicates that an access to this line should be served from the cache, even if the address is in a noncacheable segment, and that this line should never be replaced. The line set to debug_lock holds debug opcodes for breakpoints. When the swbpt bits are set to debug_lock, this value is changed only by an LPB write to the instruction cache tag memory.

When a breakpoint occurs for a line set to debug_lock, the debug host or debug monitor then sets the line to dont_use_once to enable the actual instruction to be executed instead of the debug opcode. This programming of the swbpt bits causes the instruction cache to fetch the line from main memory rather than using the contents of the cache. Once the first access to this line has been observed, the cache control logic sets the swbpt bits to

```
/*****************************************************************************/
//         Sole property of Lucent Technologies Inc. containing its
//         Proprietary Confidential information.
//
//         This unpublished source code is CONFIDENTIAL PROPRIETARY of
//                       Lucent Technologies Inc.
//
//  Project: PEGASUS
//  Block:   ic_d
//  Name:    ic_d.v
//
//  Description: This is a block in the PEGASUS SuperCore(tm). The function
//               of this block is Instruction Cache Debug Mode
//
//  Change List:
//  Date        By              Description
//  ----------  --------------  -----------------------------------------------
//  4/03/00     Han Q. Nguyen   Ver 1.0 code structure
//  6/22/00     Han Q. Nguyen   Ver 2.0 code structure with GenComps Library
//
/*****************************************************************************/
`include "../../timescale.v"

module ic_d ( gclkw, pclk, resetb, psel_id, psel_it, prdata, pwdata, cm_inval,
tag_addr,
            penable, paddr, pwrite, pwait, tag_rdata, tag_rdatb, tag_wdat,
dtag_ad,
            tag_we_a, tag_we_b, dat_rdata, dat_rdatb, dat_wdat, ddat_ad, dat_bw,
dat_we_a,
            dat_we_b, icd_stall, pmem_rd, di_bg_hip, lpbgnt, va_in, va_out,
clear_v,
            vb_in, vb_out, lrui, lruo, updf_a, updf_b, updl, upd_sb_a, upd_sb_b,
sb_i, sb_a,
            sb_b, cm_icfa, cm_icfl, cm_clva, dbg_dont_use_one, ysen1, ysen0
);

input gclkw;                    // global system clock
input pclk;             // LPB clock
input resetb;                   // global pegasus reset
input psel_id;          // Debug mode chipselect from BG
input psel_it;          // Debug mode chipselect from BG
output  [31:0]  prdata;         // LPB read data
input   [31:0]  pwdata;         // LPB write data
input   cm_inval;               // command from CM saying clear valid bit on
write
input  [31:12]  tag_addr;       // latched TAG addr
input   penable;              // LPB Enable signal
input   [31:0]  paddr;          // LPB address
input   pwrite;               // LPB write signal
output  pwait;                // LPB wait signal

input   [19:0]  tag_rdata;      // TAG Write data
input   [19:0]  tag_rdatb;      // TAG Write data
output  [19:0]  tag_wdat;       // TAG Read data
output  [6:0]   dtag_ad;       // TAG Address
output  tag_we_a;             // TAG Address
output  tag_we_b;             // TAG Address
```

```
input   [255:0]  dat_rdata;      // DATA Write data
input   [255:0]  dat_rdatb;      // DATA Write data
output  [127:0]  dat_wdat;       // DATA Read data
output  [255:0]   dat_bw;        // DATA Address
output  [6:0]   ddat_ad;        // DATA Address
output  dat_we_a;               // data write enable 1a
output  dat_we_b;               // data write enable 2b

output  icd_stall;              // stall SC when access IC in debug mode
input   pmem_rd;                // program read strobe from sc
output  lpbgnt;                 // Bus grant for LPB, use for controlling MUX
input   di_bg_hip;              // high priority signal for LPB
input   va_out;                 // Valid flag A info
input   vb_out;                 // Valid flag B info
input   lruo;                   // LRU flag  info
output  va_in;                  // Valid flag A data
output  vb_in;                  // Valid flag B data
output  lrui;                   // LRU flag data
output  updf_a;                 // update valid flag
output  updf_b;                 // update valid flag
output  updl;                   // update LRU flag
output upd_sb_a;         // update SB flag
output upd_sb_b;         // update SB flag
output   [1:0]  sb_i;            // SB input data
input   [1:0]  sb_a;             // SB output data
input   [1:0]  sb_b;             // SB output data
input  cm_icfa;                  // flush all lines (set all valid bits to zero)
input  cm_icfl;                  // flush this line (set valid bit to zero)
input  [31:0] cm_clva;           // address of 32-bit data in cache to
dump/restore
output clear_v;                 // Clear Valid flag
output dbg_dont_use_one;        // don't use once flag
input           ysen0;                  // clock control by scan
input           ysen1;                  // clock control by scan

parameter       P_IDLE = 3'b000, PWS = 3'b001, P_WAIT = 3'b010, RW_ACC = 3'b011,
RW_DONE = 3'b100;
parameter       IDLE = 2'b00, P_REQ = 2'b01, P_GNT = 2'b10, P_ACK = 2'b11;
parameter       NORM = 3'b000, USE_ONCE = 3'b001, DBG_LCK = 3'b010,
DONT_USE_ONCE = 3'b100,
           FLUSH_ALL = 3'b100, FLUSH_LINE = 3'b101;
wire     [2:0]   lpb_p_state;            // present state
reg      [2:0]   lpb_n_state;           // next state
wire     [1:0]   arb_p_state;            // present state
reg      [1:0]   arb_n_state;           // next state
wire     [2:0]   dbg_p_state;            // present state
reg      [2:0]   dbg_n_state;           // next state
wire reset;              // internal reset
wire   [31:0]  ictag;           // TAG data
reg    [31:0]  icdat;           // DAT data
reg   [255:0]   dat_bw;          // DATA Address
reg   [255:0]  dat_wdati;        // DATA Read data
wire  [255:0]  dat_rdat;         // DATA Read data
wire  [19:0]  tag_rdat;          // TAG Read data

wire  cm_icfa_d;                 // flush all lines (set all valid bits to zero)
```

```
wire  cm_icfl_d;                // flush this line (set valid bit to zero)
wire  cm_icfa_p;                // flush all lines (set all valid bits to zero)
wire  cm_icfl_p;                // flush this line (set valid bit to zero)
wire   [31:0]  zero32;          // 32 bit zero
wire   [31:0]  one32;          // 32 bit zero
wire lpb_req;                 // LPB request
wire lpb_gnt;                 // LPB grant
wire lpb_ack;                 // LPB ack
wire stall;              // stall SC
wire sel_line;                // select line L,H
wire sel_ab;                  // select way A,B
wire v_ab;               // select line L,H
wire line_a, line_b;
wire dbg_use_one;
wire dbg_lock;
wire dbg_flush_line;
wire dbg_flush_all;
wire sb_stalli, d_stall;
reg sb_stall;
wire [1:0]  sb;            // SB output data
wire    clkl_0, clkl_1, pclkl_0;
wire           qual0;                  // clock qual input
wire           qual1;                  // clock qual input

       // ------------------------------------------------------------------------
-----------------
       // Qualifier element
       // ------------------------------------------------------------------------
-----------------
       phgenDFF1 U1_phgen (.CLK(gclkw), .QUAL(qual1), .YSEN(ysen1),
.QCLK(clkl_0));
       phgenDFF0 U2_phgen (.CLK(gclkw), .QUAL(qual0), .YSEN(ysen0),
.QCLK(clkl_1));
       phgenDFF1 U3_phgen (.CLK(pclk), .QUAL(qual1), .YSEN(ysen1),
.QCLK(pclkl_0));

       // ------------------------------------------------------------------------
-----------------
       // Misc. signals
       // ------------------------------------------------------------------------
-----------------
      assign qual0 = 1'b1;
      assign qual1 = 1'b1;
      assign reset = ~resetb;
      assign zero32 = 32'b0;
      assign one32 = 32'b11111111111111111111111111111111;
      assign lpbgnt = lpb_gnt;
      assign sel_line = paddr[4];
      assign sel_ab = lruo;
      // assign sel_ab = paddr[12];
      assign v_ab = (sel_ab == 1'b1)? vb_out : va_out;
      assign sb = (sel_ab == 1'b1)? sb_b : sb_a;

       // ------------------------------------------------------------------------
-----------------
       // Arbitor signal
```

```
        // ---------------------------------------------------------------------
-----------------
      assign lpb_req = (lpb_p_state == PWS);
      assign lpb_gnt = (arb_p_state == P_GNT);
      assign lpb_ack = (lpb_p_state == RW_DONE);

        // ---------------------------------------------------------------------
-----------------
        // LPB slave Wait, LPB stall SC
        // ---------------------------------------------------------------------
-----------------
      assign pwait = (lpb_n_state == PWS || lpb_p_state == PWS || lpb_p_state ==
P_WAIT);
      assign stall = ((arb_n_state == P_GNT) || (arb_p_state == P_GNT));
        assign d_stall = stall | sb_stalli;

        DFF0R #(1) U1_DFF1 (.D(d_stall), .QCLK(clk1_1), .RB(resetb),
.Q(icd_stall));

        // ---------------------------------------------------------------------
-----------------
        // LPB slave write enable
        // ---------------------------------------------------------------------
-----------------
      assign tag_we_a = ((lpb_p_state == RW_DONE) && psel_it && pwrite &&
!sel_ab && penable && lpb_gnt);
      assign tag_we_b = ((lpb_p_state == RW_DONE) && psel_it && pwrite && sel_ab
&& penable && lpb_gnt);
      assign dat_we_a = ((lpb_p_state == RW_DONE) && psel_id && pwrite &&
!sel_ab && penable && lpb_gnt);
      assign dat_we_b = ((lpb_p_state == RW_DONE) && psel_id && pwrite && sel_ab
&& penable && lpb_gnt);

        // ---------------------------------------------------------------------
-----------------
        // Array update enable
        // ---------------------------------------------------------------------
-----------------
      assign updf_a = (lruo & dbg_use_one & !lpb_gnt) | (tag_we_a) | (line_a &
dbg_flush_line & !lpb_gnt);
      assign updf_b = (~lruo & dbg_use_one & !lpb_gnt) | (tag_we_b) | (line_b &
dbg_flush_line & !lpb_gnt);
      assign updl = tag_we_a | tag_we_b;
      assign clear_v = dbg_use_one | dbg_flush_line;

        // ---------------------------------------------------------------------
-----------------
        // mux selecting 32-bit word from data array
        // ---------------------------------------------------------------------
-----------------
        always @ ( paddr or dat_rdat)
           begin
                case ( paddr[4:2] )
                  3'b000:
                     icdat = dat_rdat[31:0];
                  3'b001:
                     icdat = dat_rdat[63:32];
```

```
                3'b010:
                   icdat = dat_rdat[95:64];
                3'b011:
                   icdat = dat_rdat[127:96];
                3'b100:
                   icdat = dat_rdat[159:128];
                3'b101:
                   icdat = dat_rdat[191:160];
                3'b110:
                   icdat = dat_rdat[223:192];
                3'b111:
                   icdat = dat_rdat[255:224];
                default:
                   icdat = dat_rdat[31:0];
             endcase
          end

        // ------------------------------------------------------------------------
-----------------
        // TAG data for LPB
        // ------------------------------------------------------------------------
-----------------
      assign dat_rdat = (paddr[12] == 1'b1)? dat_rdatb : dat_rdata;
      assign tag_rdat = (paddr[12] == 1'b1)? tag_rdatb : tag_rdata;
      assign ictag = {8'b0, sb, lruo, v_ab, tag_rdat};

        // ------------------------------------------------------------------------
-----------------
        // LPB read data.  Muxing Tag or Dat array
        // ------------------------------------------------------------------------
-----------------
      assign prdata = ((lpb_p_state == RW_DONE) && penable && psel_id)? icdat :
32'bz;
      assign prdata = ((lpb_p_state == RW_DONE) && penable && psel_it)? ictag :
32'bz;

        // ------------------------------------------------------------------------
-----------------
        // TAG write data into array
        // ------------------------------------------------------------------------
-----------------
        assign tag_wdat[19:0] = pwdata[19:0];
        assign va_in = pwdata[20];
        assign vb_in = pwdata[20];
        assign lrui = pwdata[21];

        // ------------------------------------------------------------------------
-----------------
        // mux selecting 32-bit word write data into array
        // ------------------------------------------------------------------------
-----------------
        always @ (paddr or pwdata or dat_wdati)
           begin
                case ( paddr[4:2] )
                  3'b000:
                     dat_wdati = {dat_wdati[255:32], pwdata};
                  3'b001:
```

```
                    dat_wdati = {dat_wdati[255:64], pwdata, dat_wdati[31:0]};
                 3'b010:
                    dat_wdati = {dat_wdati[255:96], pwdata, dat_wdati[63:0]};
                 3'b011:
                    dat_wdati = {dat_wdati[255:128], pwdata, dat_wdati[95:0]};
                 3'b100:
                    dat_wdati = {dat_wdati[255:160], pwdata, dat_wdati[127:0]};
                 3'b101:
                    dat_wdati = {dat_wdati[255:192], pwdata, dat_wdati[159:0]};
                 3'b110:
                    dat_wdati = {dat_wdati[255:224], pwdata, dat_wdati[191:0]};
                 3'b111:
                    dat_wdati = {pwdata, dat_wdati[224:0]};
                 default:
                    dat_wdati = 256'b0;
              endcase
        end

      assign dat_wdat = (sel_line == 1'b1)? dat_wdati[255:128] :
dat_wdati[127:0];

      // ------------------------------------------------------------------------
----------------
      // mux selecting 32-bit word BW enable
      // ------------------------------------------------------------------------
----------------
      always @ ( paddr or one32 or dat_bw)
         begin
              case ( paddr[4:2] )
                3'b000:
                   dat_bw = {dat_bw[255:32], one32};
                3'b001:
                   dat_bw = {dat_bw[255:64], one32, dat_bw[31:0]};
                3'b010:
                   dat_bw = {dat_bw[255:96], one32, dat_bw[63:0]};
                3'b011:
                   dat_bw = {dat_bw[255:128], one32, dat_bw[95:0]};
                3'b100:
                   dat_bw = {dat_bw[255:160], one32, dat_bw[127:0]};
                3'b101:
                   dat_bw = {dat_bw[255:192], one32, dat_bw[159:0]};
                3'b110:
                   dat_bw = {dat_bw[255:224], one32, dat_bw[191:0]};
                3'b111:
                   dat_bw = {one32, dat_bw[223:0]};
                default:
                   dat_bw = 256'b0;
              endcase
         end

      // ------------------------------------------------------------------------
----------------
      // LPB slave address
      // ------------------------------------------------------------------------
----------------
     assign ddat_ad = (lpb_gnt == 1'b1)? paddr[11:5] : 7'b0;
     assign dtag_ad = (lpb_gnt == 1'b1)? paddr[11:5] : cm_clva[11:5];
```

```
        // ------------------------------------------------------------------------
----------------
        // Sequential state transition LPB state machine
        // ------------------------------------------------------------------------
----------------
        DFF1R #(3) U7_DFF1 (.D(lpb_n_state), .QCLK(pclkl_0), .RB(resetb),
.Q(lpb_p_state));

        // ------------------------------------------------------------------------
----------------
        // Comb logic LPB state stranstion Block
        // ------------------------------------------------------------------------
----------------
        always @ ( lpb_p_state or lpb_gnt or psel_it or psel_id )
        begin
                case ( lpb_p_state )
                  P_IDLE:
                     begin
                        if ( (psel_it || psel_id) )
                         begin
                          if ( !lpb_gnt )
                            lpb_n_state = PWS;
                          else
                            lpb_n_state = RW_ACC;
                         end
                        else
                          lpb_n_state = P_IDLE;
                     end
                  PWS:
                        lpb_n_state = P_WAIT;
                  P_WAIT:
                    begin
                        if ( lpb_gnt )
                          lpb_n_state = RW_ACC;
                        else
                          lpb_n_state = P_WAIT;
                    end
                  RW_ACC:
                        lpb_n_state = RW_DONE;
                  RW_DONE:
                        lpb_n_state = P_IDLE;
                  default:
                        lpb_n_state = P_IDLE;
                endcase
        end

        // ------------------------------------------------------------------------
----------------
        // Sequential state transition Arbitor state machine
        // ------------------------------------------------------------------------
----------------
        DFF1R #(2) U2_DFF1 (.D(arb_n_state), .QCLK(clkl_0), .RB(resetb),
.Q(arb_p_state));

        // ------------------------------------------------------------------------
----------------
```

```
        // Comb logic Arbitor state stranstion Block
        // ---------------------------------------------------------------------
-----------------
        always @ ( arb_p_state or pmem_rd or lpb_ack or lpb_req or di_bg_hip)
        begin
                case ( arb_p_state )
                  IDLE:
                     begin
                        if ( lpb_req )
                          arb_n_state = P_REQ;
                        else
                          arb_n_state = IDLE;
                     end
                  P_REQ:
                    begin
                     if (!lpb_req)
                       begin
                        if (di_bg_hip)
                         arb_n_state = P_GNT;
                        else
                         begin
                          if (!pmem_rd)
                           arb_n_state = P_GNT;
                          else
                           arb_n_state = P_REQ;
                         end
                       end
                     else
                       arb_n_state = P_REQ;
                    end
                  P_GNT:
                    if ( lpb_ack )
                     arb_n_state = P_ACK;
                    else
                     arb_n_state = P_GNT;
                  P_ACK:
                    arb_n_state = IDLE;
                  default:
                    arb_n_state = IDLE;
                endcase
        end


        // ---------------------------------------------------------------------
-----------------
        // Sequential state transition DBG state machine
        // ---------------------------------------------------------------------
-----------------
        DFF1R #(3) U3_DFF1 (.D(dbg_n_state), .QCLK(clk1_0), .RB(resetb),
.Q(dbg_p_state));

        // ---------------------------------------------------------------------
-----------------
        // Comb logic DBG state stranstion Block
      // NORM = 3'b000, USE_ONCE = 3'b001, DBG_LCK = 3'b010, DONT_USE_ONCE =
3'b011,
      // FLUSH_ALL = 3'b100, FLUSH_LINE;
```

```
        // ------------------------------------------------------------------
----------------
        always @ ( dbg_p_state or sb or cm_icfa_p or cm_icfl_p or lpb_gnt or
cm_inval)
        begin
                case ( dbg_p_state )
                  NORM:
                     begin
                      if (!lpb_gnt)
                       begin
                        if (sb == 2'b01)
                          dbg_n_state = USE_ONCE;
                        else if (sb == 2'b10)
                          dbg_n_state = DBG_LCK;
                        else if (sb == 2'b11)
                          dbg_n_state = DONT_USE_ONCE;
                        else if (cm_icfl_p | cm_inval)
                          dbg_n_state = FLUSH_LINE;
                        else if (cm_icfa_p)
                          dbg_n_state = FLUSH_ALL;
                        else
                          dbg_n_state = NORM;
                       end
                      else
                        dbg_n_state = NORM;
                     end
                  USE_ONCE:
                    begin
                          dbg_n_state = NORM;
                    end
                  DBG_LCK:
                          if ( sb == 2'b00 )
                            dbg_n_state = NORM;
                          else
                            dbg_n_state = DBG_LCK;
                  DONT_USE_ONCE:
                            dbg_n_state = DBG_LCK;
                  FLUSH_ALL:
                            dbg_n_state = NORM;
                  FLUSH_LINE:
                            dbg_n_state = NORM;
                  default:
                          dbg_n_state = NORM;
                endcase
        end

        // ------------------------------------------------------------------
----------------
        // Comparator for tag match
        // ------------------------------------------------------------------
----------------
      assign line_a = ((tag_rdata[19:0] == cm_clva[31:12]) ||
                  (tag_rdata[19:0] == tag_addr[31:12]))? 1'b1 : 1'b0;
      assign line_b = ((tag_rdatb[19:0] == cm_clva[31:12]) ||
                  (tag_rdata[19:0] == tag_addr[31:12]))? 1'b1 : 1'b0;
```

```
        // ------------------------------------------------------------------------
----------------
        //  control logics for debug SB states
        // ------------------------------------------------------------------------
----------------
      assign dbg_use_one = (dbg_p_state == USE_ONCE);
      assign dbg_lock = (dbg_p_state == DBG_LCK);
      assign dbg_dont_use_one = (dbg_p_state == DONT_USE_ONCE);
      assign dbg_flush_line = (dbg_p_state == FLUSH_LINE);
      assign dbg_flush_all = (dbg_p_state == FLUSH_ALL);


        // ------------------------------------------------------------------------
----------------
        //  stall SC for clearing SB bits in use-Once mode
        // ------------------------------------------------------------------------
----------------
      assign sb_stalli = (dbg_n_state == USE_ONCE || dbg_n_state == FLUSH_LINE
|| dbg_n_state == FLUSH_ALL);

        // ------------------------------------------------------------------------
----------------
        // Mux for SB bits
        // sb_i[1:0] = pwdata[23:22];
        // ------------------------------------------------------------------------
----------------
      reg   [1:0]  sb_i;           // SB input data
        always @ (lpb_gnt or dbg_use_one or dbg_dont_use_one or pwdata)
        begin
                case ( {lpb_gnt, dbg_dont_use_one, dbg_use_one} )
                  3'b001:
                    sb_i[1:0] = 2'b00;
                  3'b010:
                    sb_i[1:0] = 2'b10;
                  3'b100:
                    sb_i[1:0] = pwdata[23:22];
                  default:
                    sb_i[1:0] = 2'b00;
                endcase
        end

        // ------------------------------------------------------------------------
----------------
        // Update SB array
        // ------------------------------------------------------------------------
----------------
      assign upd_sb_a = (dbg_use_one & !lpb_gnt & lruo) | (dbg_dont_use_one &
!lpb_gnt & lruo) | tag_we_a;
      assign upd_sb_b = (dbg_use_one & !lpb_gnt & !lruo) | (dbg_dont_use_one &
!lpb_gnt & !lruo) | tag_we_b;


        // ------------------------------------------------------------------------
----------------
        // flush pulse
        // ------------------------------------------------------------------------
----------------
```

```
        DFF1R #(1) U4_DFF1 (.D(cm_icfa), .QCLK(clkl_0), .RB(resetb),
.Q(cm_icfa_d));

        DFF1R #(1) U5_DFF1 (.D(cm_icfl), .QCLK(clkl_0), .RB(resetb),
.Q(cm_icfl_d));

      assign cm_icfa_p = cm_icfa & ~cm_icfa_d;
      assign cm_icfl_p = cm_icfl & ~cm_icfl_d;




endmodule
```

```
# Directory of: sync://star.agere.com:2645/Projects/Peg/chip/pegIP/ic/src/ic_d.v;
#
# fo---      9/07/2000 13:58:38 ic_d.v;1.1


import -version 1.1 sync://star.agere.com:2645/Projects/Peg/chip/pegIP/ic/src
ic_d.v
#*  import -version 1.1 sync://star.agere.com:2645/Projects/Peg/chip/pegIP/ic/src
ic_d.v
# ic_d.v:   Success Imported
```